Department I - C Plus Plus

Modern and Lucid C++
for Professional Programmers

Week 14 – Exam Infos and Outlook

Cevelop
Your C++ deserves it

Thomas Corbat / Felix Morgner
Rapperswil, 15.12.2020
HS2020

OST
Ostschweizer
Fachhochschule

IFS
INSTITUTE FOR
SOFTWARE

Goals

- **You know what to expect from the exam**
- **You gain an overview of what C++ >=20 holds in store**
- **You are absolutely hyped to take C++ Advanced ☺**

Topics

- **Recap Week 13**
- **Exam Information**
- **Overview of C++20 and beyond**
- **Concepts**
- **Ranges**

Recap Week 13

Cevelop

Your C++ deserves it

OST Ostschweizer Fachhochschule

IFS INSTITUTE FOR SOFTWARE

Summary

- **Default Initialization**
- **Value Initialization**
- **Direct Initialization**
- **Copy Initialization**
- **List Initialization**
- **Aggregate Initialization**

- **The kind depends on the context**
  - Four general syntaxes

1. **Nothing**
2. `( expression list )`
3. `= expression`
4. `{ initializer list }`

Default Initialization

- **Danger lurks!**
  - Reading an uninitialized value incurs undefined behavior!

```
void print_uninitialized() {
  int my_number;
  std::cout << my_number << '\n';
}
```

Direct Initialization

- **Similar to Value Initialization**
  - Uses non-empty ( ) or { }
  - When using { } only applies if not a class type (see List Initialization)
- **“Most vexing parse” lurks with ( )**
  - Prefer { … }

```
#include <string>

void diri_function() {
  int number{32};
  std::string text("CPl");
  word vexing (std::string());
}
```

Most Vexing Parse

- **Two interpretations**
  - Initializition with a value-initialized string
  - Declaration of a function returning word and taking an unnamed pointer to a function returning a string
- **The first is what we would expect**
- **The second is the one the standard requires!**
  - Therefore, prefer { … }

```
word vexing (std::string());
```

Aggregates

- **Simple class types**
  - Can have other types as public base classes
  - Can have member variables and functions
  - Must not have user-provided, inherited or explicit constructors
  - Must not have protected or private direct members
- **Mostly used for “simple” types**
  - No invariant that has to be established
  - Example: DTOs
- **Arrays are also Aggregates**

Exam Information

Cevelop
Your C++ deserves it

OST Ostschweizer Fachhochschule

IFS INSTITUTE FOR SOFTWARE

Exam Information

- **3. February 2021, 12:30 – 14:30 (2h), in the Aula (4.101)**
- **Open Book**
  - All exercises, lectures, notes, books, standards, memes, …
  - **Absolutely no old exams or excerpts from old exams!! (we WILL check!!)**
- **What we will expect from you:**
  - Code comprehension (reading and understanding)
  - Code production (writing)
  - Theoretical knowledge
- **DON’T PANIC!**

Recommended Resources

- **CPP Quiz (https://cppquiz.org)**
- **CpI Lecture Videos and Slides**
- **Exercises**
  - Including Testat Feedback!
- **C++ reference (https://cppreference.com)**
- **Teamwork!**

C++ >=20 and CplA

- **C++ 20 is the biggest release since C++ 11**
  - Coroutines
  - **Constraints and Concepts**
  - **Ranges**
  - Template Lambdas
  - 3-way Comparison (`operator<=>`)
  - First steps towards reflection
  - String Formatting (fmt)
  - Shorter syntax for function templates
  - …

- **Work on C++23 is already in progress**
  - Stacktrace support
  - String extensions
  - Additional numeric literals
- **Features we are hopeful about being included**
  - Contracts
  - Networking and async I/O
  - Static Reflection

- **Topics (most-likely) covered in CpIA**
  - Explicit Heap Memory Management
  - Custom Iterators
  - Compile-time Computation
  - Multithreading
  - Networking and async I/O
  - Build Systems
  - “Bring Your Own Topic”

Concepts

Cevelop

Your C++ deserves it

OST Ostschweizer Fachhochschule

IFS INSTITUTE FOR SOFTWARE

Concepts

- **Concept: The requirements a type must fulfill to be useable as an argument for a specific template parameter**

- **What are the requirements of the type `T` in our `min` function template?**

```
template <typename T>
T min(T left, T right) {
  return left < right ? left : right;
}
```

  - < Comparable with itself: `bool operator<(T, T)`
  - Copy/Move-Constructible, to return T by value

- **In C++20 it is possible to explicitly specify concepts**
  - Allows better checking of template definition (are all requirements fulfilled)
  - Better (easier to read) error messages for failed template instantiations

Expressing concepts in C++ <20

- **Before C++ 20, Template Metaprogramming was used**
  - Often based on `<type_traits>`
  - Hard to get right and understand
  - Almost impossible to “debug”

```
template<typename T>
auto constexpr is_lessthan_comparable =
  std::is_same_v<bool, decltype(std::declval<T>() < std::declval<T>())>;
```

Applying concepts in C++ <20

- **Before C++ 20, Template Metaprogramming was used**
  - Often based on `std::enable_if`
  - Allows different overloads using **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror (**SFINAE**)
  - Still ugly error messages

```
template<typename T>
std::enable_if_t<is_lessthan_comparable<T>, T> min(T left, T right) {
  return left < right ? left : right;
}
```

Expressing concepts in C++ >=20

- **C++ 20 adds explicit syntax for concepts**
  - Much cleaner syntax
  - The compiler understands that something is a concept
  - Large number of concepts in `<concepts>` and `<iterator>`
  - New `concept` keyword

```cpp
template<typename T>
concept lessthan_comparable = requires(T a, T b) {
    { a < b } -> std::same_as<bool>;
};
```

Applying Concepts in C++ >=20

- **C++ 20 add a clean syntax to apply concepts**
  - Via `requires` clauses

```
template<typename T>
T min(T left, T right) requires lessthan_comparable<T> {
  return left < right ? left : right;
}
```

  - Or instead of `typename`

```
template<lessthan_comparable T>
T min(T left, T right) {
  return left < right ? left : right;
}
```

Applying Concepts in C++ >=20

- **This provides us with useful error messages**

```
..\main.cpp:36:41: error: use of function 'T min(T, T) [with T = blubber]' with unsatisfied
constraints
   36 |  auto smaller = min(blubber{}, blubber{});
      |                                         ^
..\main.cpp:30:3: note: declared here
   30 | T min(T left, T right) {
      |   ^~~
..\main.cpp:30:3: note: constraints not satisfied
..\main.cpp: In instantiation of 'T min(T, T) [with T = blubber]':
..\main.cpp:36:41:   required from here
..\main.cpp:16:9:   required for the satisfaction of 'lessthan_comparable<T>' [with T = blubber]
..\main.cpp:16:31:   in requirements with 'T a', 'T b' [with T = blubber]
..\main.cpp:17:9: note: the required expression '(a < b)' is invalid, because
   17 |     { a < b } -> std::same_as<bool>;
      |       ~~^~~
..\main.cpp:17:9: error: no match for 'operator<' (operand types are 'blubber' and 'blubber')
```

Ad-hoc Requirements

- Requirements can also be specified ad-hoc

```
template<typename T>
T min(T left, T right) requires requires (T a, T b){ a < b; } {
  return left < right ? left : right;
}
```

- While somewhat limited, can be useful for one-off constraints

Overloading with Concepts

- **Concepts can be used to select specific overloads**

```
template<lessthan_comparable T>
T min(T left, T right) {
  return left < right ? left : right;
}

template<greaterthan_comparable T>
T min(T left, T right) {
  return right > left ? left : right;
}
```

- **Useful when specializing algorithm implementations**
  - `iterator_traits`

Concepts Summary

- **Concepts allow us to express the requirements on a type to the compiler**
  - The compiler understands that something is a concept
  - Allows for better error messages
- **Requirements can be formulated ad-hoc**
  - Quick one-off constraints
- **Functions can be overloaded based on concepts**

Ranges

Ranges in C++ <20

- **Many algorithms work on one or more “ranges”**
- **A range is a pair of iterators**
  - Usually begin and end
- **Think back to copy:**

```
int main() {
  std::vector numbers{1, 2, 3, 4};
  copy(cbegin(numbers), cend(numbers), std::ostream_iterator<int>(std::cout));
}
```

- **This is boring and cumbersome**
  - And somewhat violates the DRY principle in most cases

Ranges in C++ >=20

- **C++ 20 add the ranges library**
  - New “variants” of all algorithms (except numeric)
  - Views
    - Allowing for filtering, transformations, etc.
  - All new functions are defined in `std::ranges` and `std::views`
- **All standard containers are ranges**
- **More basic, everything that has `begin` and `end` is a range**
  - Plus some additional requirements, expressed via concepts

Simplifying copy

```
int main() {
  std::vector numbers{1, 2, 3, 4};
  std::ranges::copy(numbers, std::ostream_iterator<int>(std::cout));
}
```

Ranges and Views

- **In addition to ranges, the library provides views**
  - views::take_while
  - views::drop_while
  - views::reverse
  - …
- **Views can be created from and combined with ranges**

```
int main() {
  std::vector numbers{1, 2, 3, 4};
  auto reversed = std::ranges::reverse_view(numbers);
  std::ranges::copy(reversed, std::ostream_iterator<int>(std::cout));
}
```

View Pipelines

- **Views can also be combined into “pipelines”**
  - E.g.: print the third-power of all odd numbers in a vector, ignoring the first one

```
int main() {
  auto odd = [](auto n){ return n % 2; };
  auto pow3 = [](auto n) { return n * n * n; };

  std::vector numbers{1, 2, 3, 4};
  auto transformed = numbers |
    std::views::filter(odd) |
    std::views::transform(pow3) |
    std::views::drop(1);
  std::ranges::copy(transformed, std::ostream_iterator<int>(std::cout));
}
```

Ranges Summary

- **Ranges reduce the amount of code we have to write**
- **They allow us to define new sub- or transformed ranges**

CPI Summary

- **C++ is a powerful language**
  - It provides different paradigms (OOP, Functional, Structured, …)
- **Modern C++ is expressive and (mostly) clean**
- **C++ allows the implementation of high-performance programs on a high level of abstraction**
- **The language keeps evolving**
  - C++ 20 being the largest release since C++ 11
  - YOU could participate by writing and or reviewing proposals